CS202 (003): Operating Systems
File System V

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Last Time

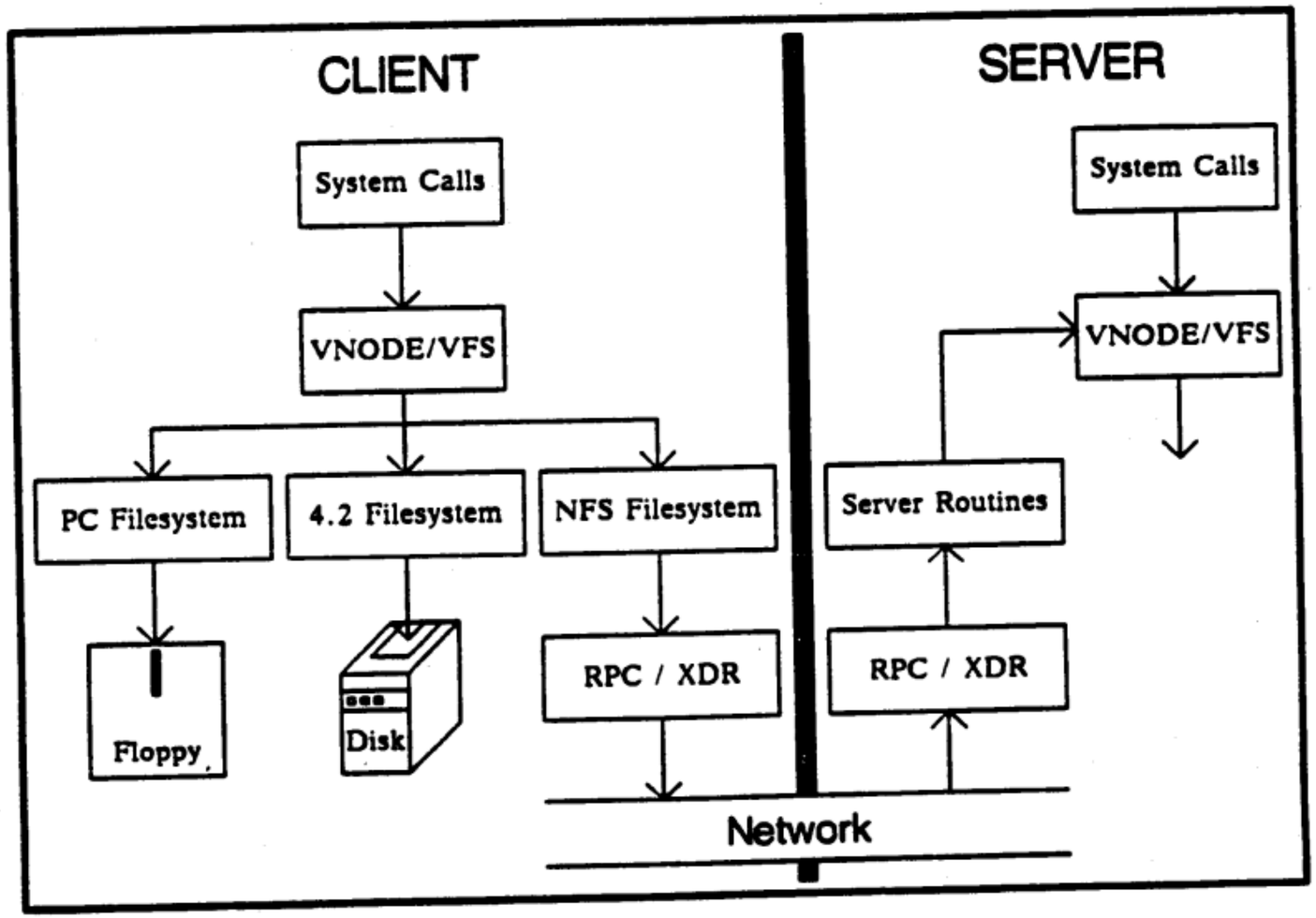

Figure 1

Transparency

Transparency requires that the system calls **mean** the same things

Gen #

What if client A deletes a file and it (or another client) creates a new one that uses the same i-node?

The server maintains a generation number in each i-node on disk
Every time an i-node is reallocated (used for a new file), its generation number is incremented
When a client gets a file handle (FH) through operations like LOOKUP, the current generation number is included in that file handle

For every client request, the server compares two numbers:
1. The generation number in the client's file handle
2. The current generation number stored in the i-node on disk
If they match: The request is valid and proceeds normally
If they don't match: The client gets a "stale FH" error when trying to READ() or WRITE()

Non-traditional Unix Semantics

Error returns on successful operations

Non-traditional Unix Semantics

Close-to-open consistency

When client A writes and close a file, Client B will only see those changes after opening the file

Non-traditional Unix Semantics

Close-to-open consistency

Server must flush to the disk before returning

The server has to make sure, before returning:

1. Inode with new block # and new length safe on disk
2. Indirect block safe on disk

Writes have to be synchronous

Would this case performance issue?

Non-traditional Unix Semantics

Would this case performance issue?

No, because there are caching (at the client; not all RPCs go to server. although write go to the server in NFSv3, they don't cause disk accesses necessarily)

Read-caching
(useful when re-reading files)

Write-caching
(improve performance)

Caching of file attributes
(helps with command such as `ls -l`)

Caching of name->fh mapping
(Caches path prefixes (e.g., /home/jo))

But, now you have to worry about coherence and semantics!

Close-to-open consistency

When client A writes and close a file, Client B will only see those changes after opening the file

Non-traditional Unix Semantics

Close-to-open consistency

When client A writes and close a file, Client B will only see those changes after opening the file

1. writing client forces dirty blocks during a close()
2. reading client checks with server during open(): “is this data current?”

Hmmm, why not a stronger guarantee?

Trading stronger guarantee for better performance!

Obviously, this might cause issues, for example:

1. Errors might occur on close() rather than write()
2. Legacy applications that don't check close() return values might fail
3. Certain usage patterns don't work well, such as using "tail -f" on one client while another client writes to the file

Non-traditional Unix Semantics

Server failure

Previously: open(“some_file”, RD_ONLY) failed if “some_file” does not exist
Now: app might hang while trying to access the file

Deletion or permission change of open files

What is Client A deletes a file that Client B has “open”?

Previously: Client B reads still work (file exists until all clients close() it)
Now: Client B reads fail

What is Client A make the file inaccessible to others while Client B has the file open()?

Previously: Nothing happens
Now: Client B reads fail

……

Security

NFS's only security measure is IP address verification (which is quite weak)

Previously: Unix enforces read/write protections — cannot read my files w/o passwords
Now: Server believes whatever UID appears in NFS request (and anyone can put whatever in the request)

Not extremely vulnerable because of how FH works

```
Example structure (simplified):
struct file_handle {
    uint32_t filesystem_id;     // Random unique identifier
    uint32_t inode_number;      // File system location
    uint32_t generation_number; // Changes when inode is reused
    uint8_t  extra_data[20];    // Additional metadata
}
```

It does not solve all types of attack though!

Vulnerabilities are technically fixable (strong auth, secure protocols, …),
but hard to reconcile with the stateless design